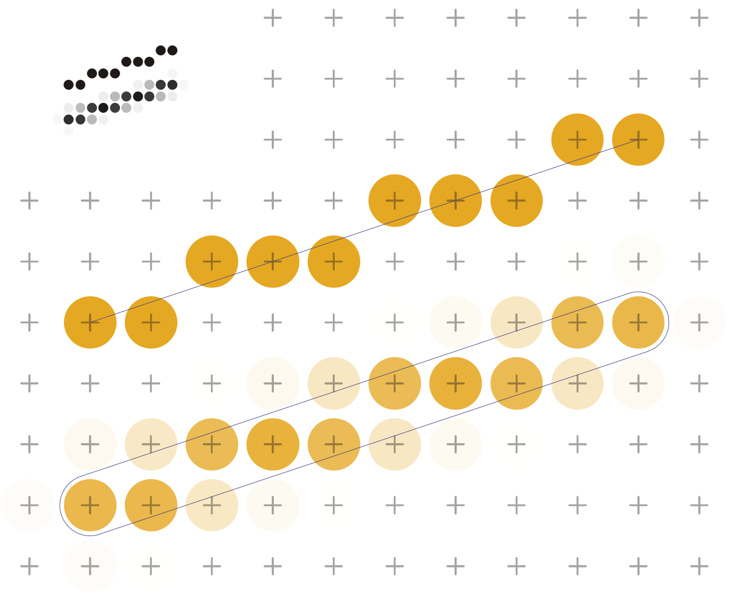


Scan Conversion

Andries van Dam

Scan Converting Lines

Line Drawing

- Draw a line on a raster screen between two points
- Why is this a difficult problem?
  - What is “drawing” on a raster display?
  - What is a “line” in raster world?
  - Efficiency and appearance are both important

Problem Statement

- Given two points P and Q in XY plane, both with integer coordinates, determine which pixels on raster screen should be on in order to draw a unit-width line segment starting at P and ending at Q

What is Scan Conversion?

- Final step of rasterisation (process of taking geometric shapes and converting them into an array of pixels stored in the framebuffer to be displayed)
- Takes place after clipping occurs
- All graphics packages do this at the end of the rendering pipeline
- Takes triangles and maps them to pixels on the screen
- Also takes into account other properties like lighting and shading, but we'll focus first on algorithms for line scan conversion

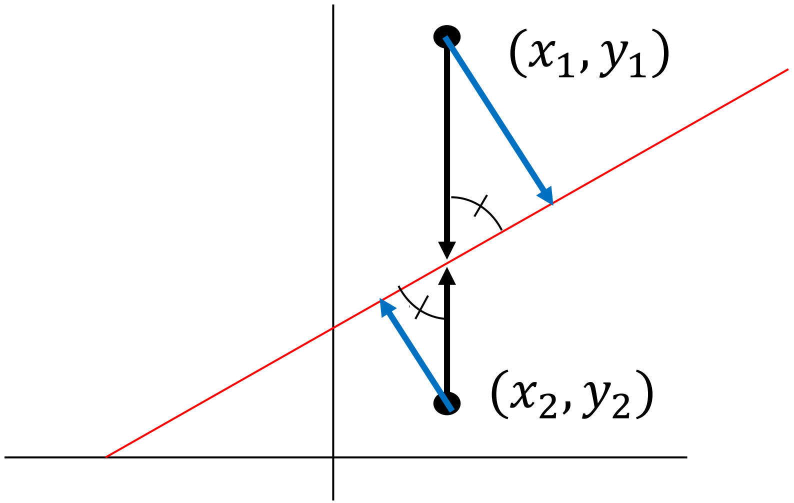
Finding the next pixel:

Special cases:

- **Horizontal Line:**
  - Draw pixel P and increment x coordinate value by 1 to get next pixel.
- **Vertical Line:**
  - Draw pixel P and increment y coordinate value by 1 to get next pixel.
- **Diagonal Line:**
  - Draw pixel P and increment both x and y coordinate by 1 to get next pixel.
- What should we do in general case?
  - Increment x coordinate by 1 and choose point closest to line.
  - But how do we measure "closest"?

Vertical Distance

- Why can we use vertical distance as a measure of which point is closer?
  - ... because vertical distance is proportional to actual distance
- Similar triangles show that true distances to line (in blue) are directly proportional to vertical distances to line (in black) for each point
- Therefore, point with smaller vertical distance to line is closest to line



Strategy 1 – Incremental Algorithm (1/3)

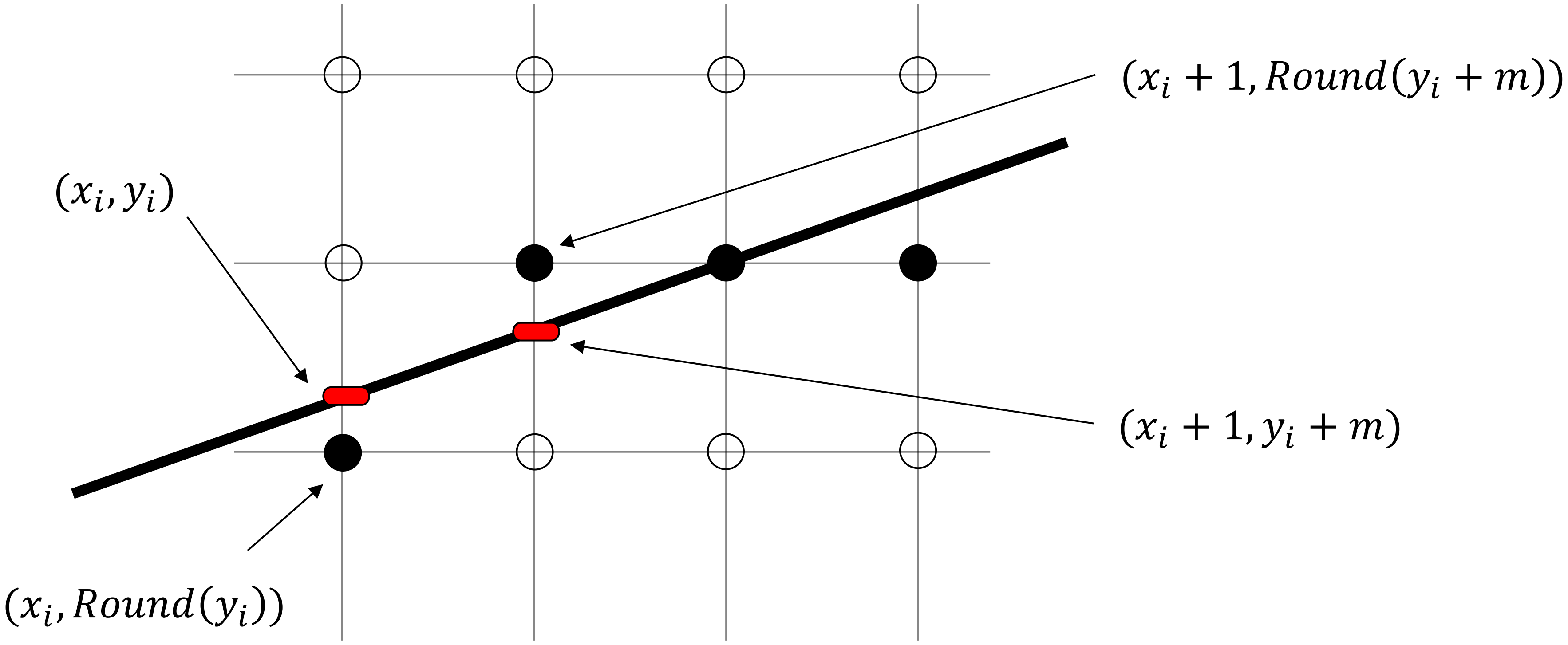
Basic Algorithm:

- Find equation of line that connects two points P and Q
- Starting with leftmost point , increment x_i by 1 to calculate $y_i = m * x_i + B$ where m = slope, B = y intercept
- Draw pixel at $(x_i, \text{Round}(y_i))$ where Round $(y_i) = \lfloor .5 + y_i \rfloor$

Incremental Algorithm:

- Each iteration requires a floating-point multiplication
  - Modify algorithm to use deltas
  - $(y_{i+1} - y_i) = m * (x_{i+1} - x_i)$
  - $y_{i+1} = y_i + m * (x_{i+1} - x_i)$
  - If $\Delta x = x_{i+1} - x_i = 1$, then $y_{i+1} = y_i + m$
- At each step, we make incremental calculations based on preceding step to find next y value

Strategy 1 – Incremental Algorithm (2/3)



Sample Code and Problems (3/3)

```
void Line(int x0, int y0, int x1, int y1) {
    int   x, y;
    float dy = y1 – y0;
    float dx = x1 – x0;
    float m  = dy / dx;

    y = y0;
    for (x = x0; x < x1; ++x) {
        WritePixel( x, Round(y) );
        y = y + m;
    }
}
```

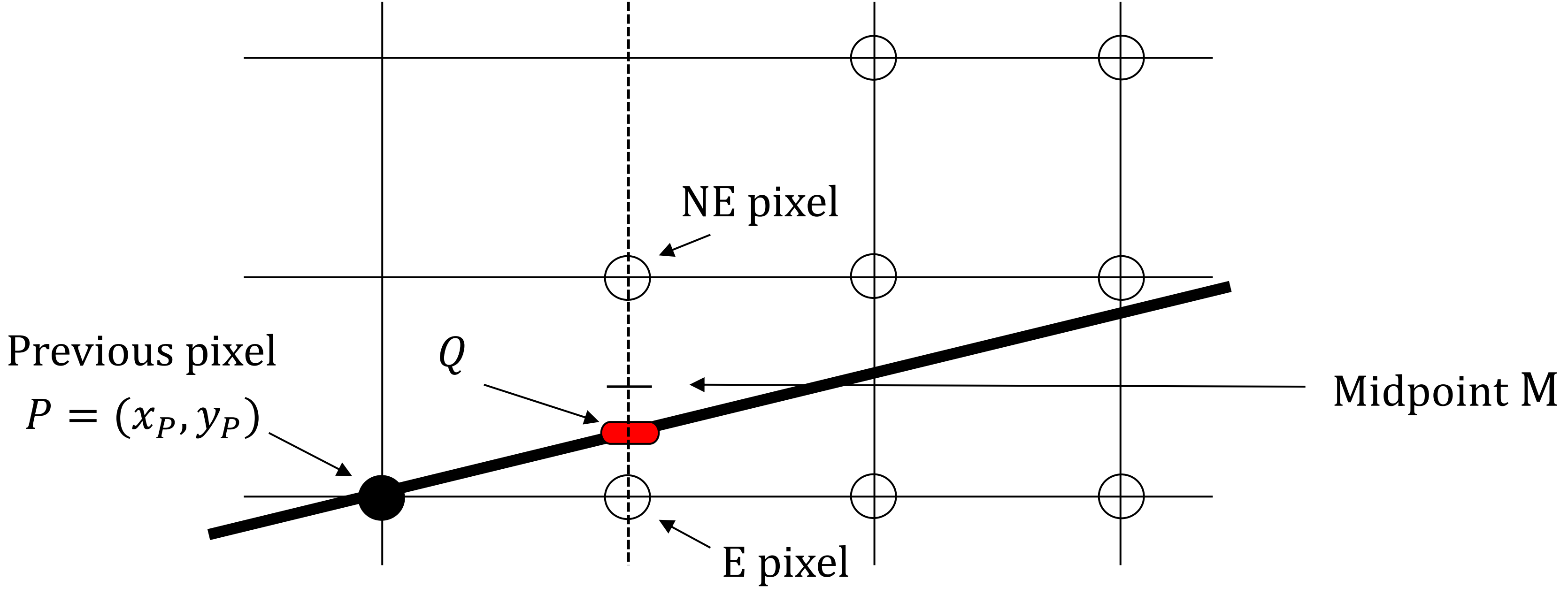
Since slope is fractional, need special case for vertical lines (dx = 0)

Rounding takes time

Strategy 2 – Midpoint Line Algorithm (1/3)

- Assume that line's slope is shallow and positive (0 < slope < 1); other slopes can be handled by suitable reflections about principle axes
- Call lower left endpoint (x_0, y_0) and upper right endpoint (x_1, y_1)
- Assume that we have just selected pixel P at (x_P, y_P)
- Next, we must choose between pixel to right (E pixel), or one right and one up (NE pixel)
- Let Q be intersection point of line being scan-converted and vertical line $x = x_P + 1$

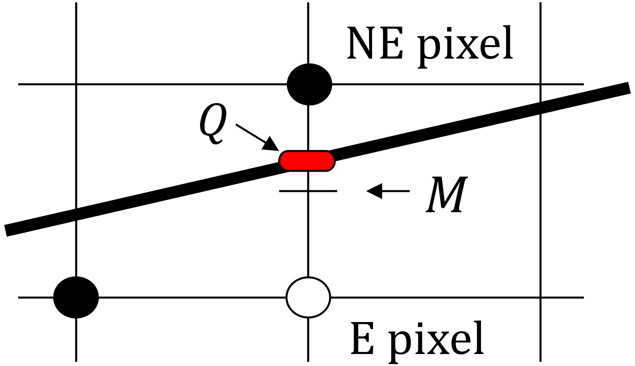
Strategy 2 – Midpoint Line Algorithm (2/3)



$x = x_P + 1$ (dashed)

Strategy 2- Midpoint Line Algorithm (3/3)

- Line passes between E and NE
- Point that is closer to intersection point Q must be chosen
- Observe on which side of line midpoint M lies:
  - E is closer to line if midpoint M lies above line, i.e., line crosses bottom half
  - NE is closer to line if midpoint M lies below line, i.e., line crosses top half
- Error (vertical distance between chosen pixel and actual line) is always $\leq .5$



For line shown, algorithm chooses NE as next pixel.

Now, need to find a way to calculate on which side of line midpoint lies

General Line Equation

- Line equation as function: $f(x) = y = mx + B = \frac{dy}{dx}x + B$
- Line equation as implicit function: $f(x, y) = ax + by + c = 0$
  - Avoids infinite slopes, provides symmetry between x and y
- So from above,

$$y \cdot dx = dy \cdot x + B \cdot dx$$
$$dy \cdot x - y \cdot dx + B \cdot dx = 0$$
$$\therefore a = dy, b = -dx, c = B \cdot dx$$

- Properties (proof by case analysis):
  - $f(x_m, y_m) = 0$ when any point m is on line
  - $f(x_m, y_m) < 0$ when any point m is above line
  - $f(x_m, y_m) > 0$ when any point m is below line
  - Our decision will be based on value of function at midpoint M at $(x_P + 1, y_P + .5)$

Decision Variable

Decision Variable d :

- We only need sign of $f(x_P + 1, y_P + .5)$ to see where the line lies, and then pick nearest pixel.
- $d = f(x_P + 1, y_P + .5)$
  - if $d > 0$ choose pixel NE
  - if $d < 0$ choose pixel E
  - if $d = 0$ choose either one consistently

How do we incrementally update d?

- On basis of picking E or NE, figure out location of M for the next pixel, and corresponding value d for next grid line.
- We can derive d for the next pixel based on our current decision.

Incrementing Decision Variable if E was chosen:

Increment *M* by one in *x* direction:

- $d_{old} \;= a(x_P + 1) + b(y_P + .5) + c$
- $d_{new} = f(x_P + 2, y_P + .5)$
  $= a(x_P + 2) + b(y_P + .5) + c$
- $d_{new} - \; d_{old}$ is the incremental difference ΔE
  - $d_{new} = \; d_{old} + a \rightarrow \Delta E = a = dy$ (2 slides back)
- We can compute value of decision variable at next step incrementally without computing *F(M)* directly
  - $d_{new} = \; d_{old} + \Delta E = d_{old} + dy$
- ΔE can be thought of as correction or update factor to take d_{old} to d_{new}
- It is referred to as <u>forward difference</u>

If NE was chosen:

Increment M by one in both *x* and *y* directions:

- $d_{new} = f(x_P + 2, y_P + 1.5)$

  $= a(x_P + 2) + b(y_P + 1.5) + c$
- $\Delta\text{NE} = d_{new} - d_{old}$
  - $d_{new} = d_{old} + a + b \rightarrow \Delta\text{NE} = a + b = dy - dx$
- Thus, incrementally,

  $d_{new} = d_{old} + \Delta\text{NE} = d_{old} + dy - dx$

Summary (1/2)

- At each step, algorithm chooses between 2 pixels based on sign of decision variable calculated in previous iteration.
- It then updates decision variable by adding either ΔE or ΔNE to old value depending on choice of pixel. Simple additions only!
- First pixel is first endpoint (x_0, y_0), so we can directly calculate initial value of d for choosing between E and NE.

Summary (2/2)

- First midpoint for first $d = d_{start}$ is at $(x_0 + 1, y_0 + .5)$
- f $(x_0 + 1, y_0 + .5)$

  $= a(x_0 + 1) + b(y_0 + .5) + c$

  $= ax_0 + by_0 + a + \frac{b}{2} + \text{c}$

  $= \text{f}\,(x_0, y_0) + a + \frac{b}{2}$
- But (x_0, y_0) is point on line, so f $(x_0, y_0) = 0$
- Therefore, $d_{start} = a + \frac{b}{2} = dy - \frac{dx}{2}$
  - use d_{start} to choose second pixel, etc.
- To eliminate fraction in d_{start}:
  - redefine f by multiplying it by 2; $f(x, y) = 2(ax + by + c)$
  - This multiplies each constant and decision variable by 2, but does not change sign
- Note: this is identical to "Bresenham's algorithm", though derived by different means. That won't be true for circle and ellipse scan conversion.

Example Code

```
void MidpointLine(int x0, int y0, int x1, int y1) {
    int dx = (x1 - x0), dy = (y1 - y0);
    int d = 2 * dy - dx;
    int incrE = 2 * dy;
    int incrNE = 2 * (dy - dx);
    int x = x0, y = y0;
    WritePixel(x, y);

    while (x < x1) {
        if (d <= 0) d = d + incrE;    // East Case
        else { d = d + incrNE; ++y; } // Northeast Case
        ++x;
        WritePixel(x, y);
    }
}
```

Scan Converting Circles

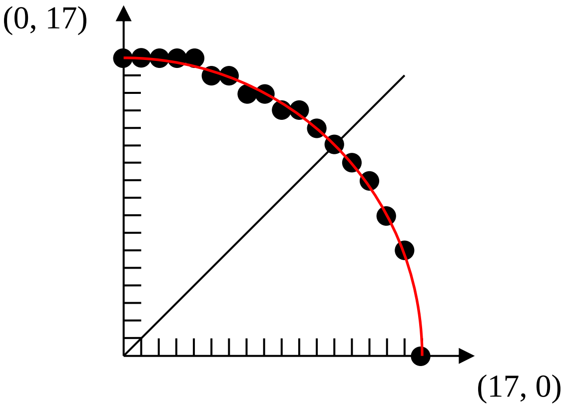
Version 1: <u>really</u> bad

For x from $-R$ to R:

$y = \sqrt{R^2 - x^2}$;

WritePixel(round(x), round(y));

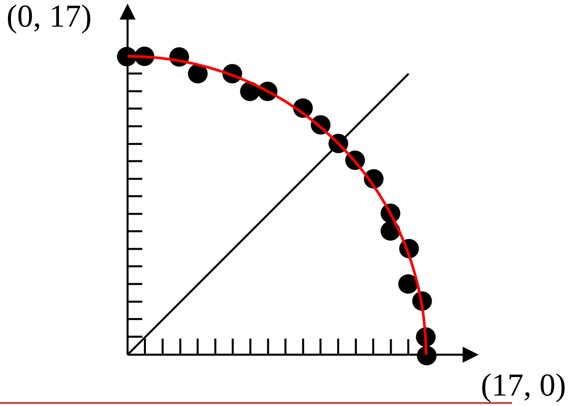
WritePixel(round(x), round($-y$));



Version 2: slightly less bad

For x from 0 to 360:

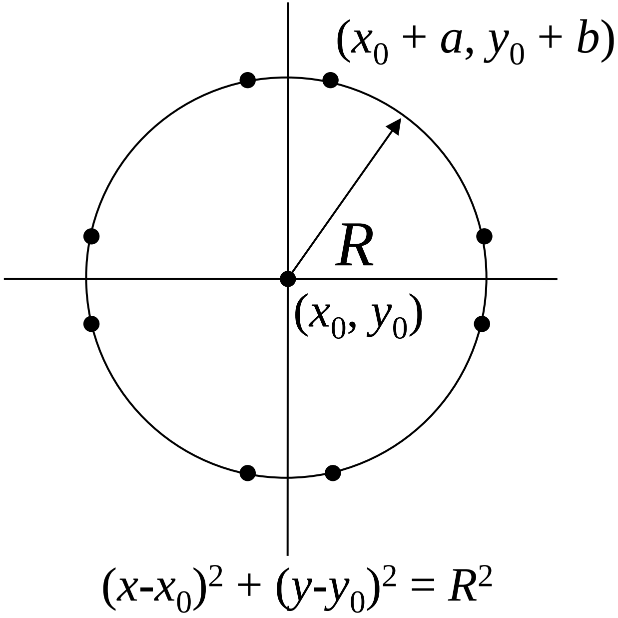
WritePixel(round($R\cos(x)$), round($R\sin(x)$));



Version 3 — Use Symmetry

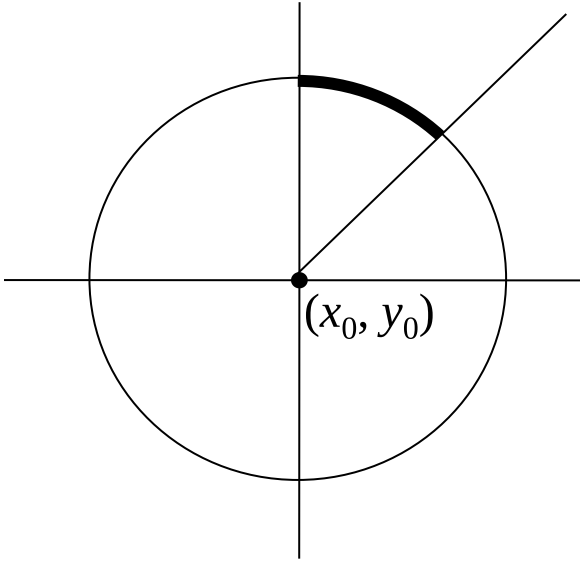
Symmetry:

- If $(x_0 + a, y_0 + b)$ is on circle centered at (x_0, y_0):
  - Then $(x_0 \pm a, y_0 \pm b)$ and $(x_0 \pm b, y_0 \pm a)$ are also on the circle
  - Hence there is 8-way symmetry
- Reduce the problem to finding the pixels for 1/8 of the circle.



Using the Symmetry

- Scan top right 1/8 of circle of radius R
- Circle starts at $(x_0, y_0 + R)$
- Let's use another incremental algorithm with decision variable evaluated at midpoint

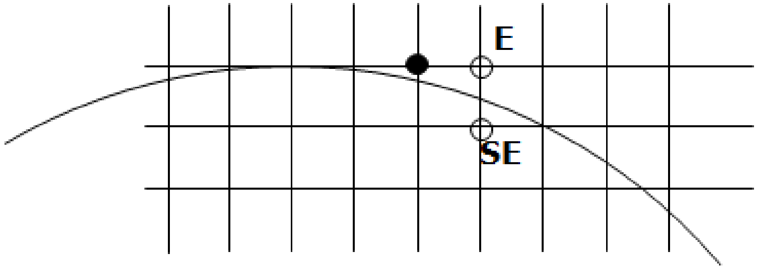


The incremental algorithm – a sketch

```
x = x0, y = y0 + R; WritePixel(x, y);

for (x = x + 1; (x - x0) < (y - y0); x++) {
     if (decision_var < 0) {
          // move east
          update decision variable
     } else {
          // move south east
          update decision variable
          y--;
     }

     WritePixel(x, y);
}
```



Note: can replace all occurrences of x_0, y_0 with 0, shifting coordinates by $(-x_0, -y_0)$

What we need for the Incremental Algorithm

- Decision variable
  - negative if we move E, positive if we move SE (or vice versa).
- Follow line strategy: Use implicit equation of circle
  - $f(x, y) = x^2 + y^2 - R^2 = 0$
  - $f(x, y)$ is zero on circle, negative inside, positive outside
- If we are at pixel (x, y) examine $(x + 1, y)$ and $(x + 1, y - 1)$
- Compute f at the midpoint.

The Decision Variable

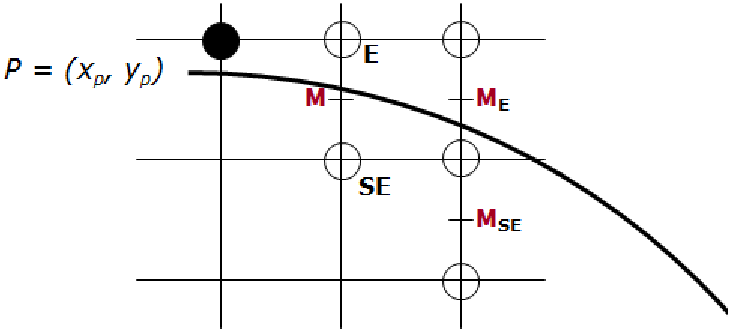
- Evaluate $f(x,y) = x^2 + y^2 - R^2$ at the point:

$$\left(x+1, y-\frac{1}{2}\right)$$

- We are asking: "Is $f(M)$ =

$$f\left(x+1, y-\frac{1}{2}\right) = (x+1)^2 + (y-\frac{1}{2})^2 - R^2$$

positive or negative?" (it is zero on circle)



- If **negative**, midpoint inside circle, **choose E**
  - *vertical* distance to the circle is less at $(x+1, y)$ than at $(x+1, y-1)$
- If **positive**, opposite is true, **choose SE**

The right decision variable?

- Decision based on vertical distance
- Ok for lines, since d and d_{vert} are proportional
- For circles, not true:

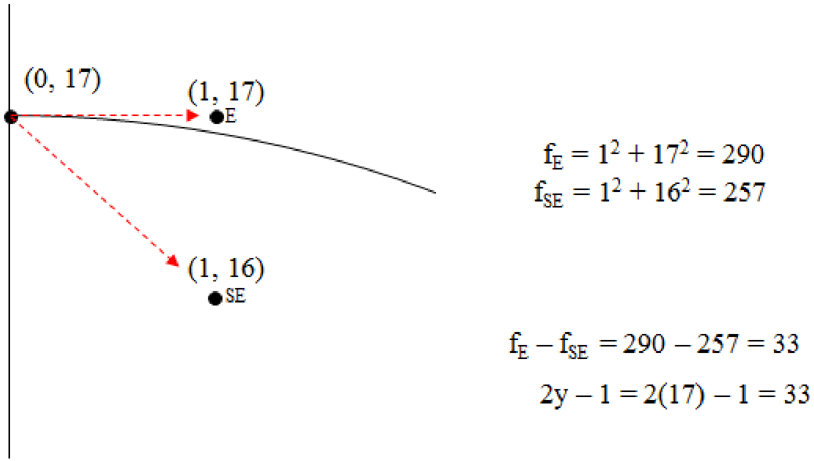
$$d\big((x+1,y),Circ\big) = \sqrt{(x+1)^2 + y^2} - R$$
$$d\big((x+1,y-1),Circ\big) = \sqrt{(x+1)^2 + (y-1)^2} - R$$

- Which d is closer to zero? (i.e., which value below is closest to R?):

$$\sqrt{(x+1)^2 + y^2} \text{ or } \sqrt{(x+1)^2 + (y-1)^2}$$

Alternate Phrasing (1/3)

- We could ask instead: "Is $(x+1)^2 + y^2$ or $(x+1)^2 + (y-1)^2$ closer to R^2?"
- The two values in equation above differ by:
- $[(x+1)^2 + y^2] - [(x+1)^2 + (y-1)^2] = 2y - 1$



Alternate Phrasing (2/3)

- The second value, which is always less, is *closer* if its difference from R^2 is less than: $\frac{1}{2}(2y-1)$

i.e., if $$R^2 - [(x+1)^2 + (y-1)^2] < \frac{1}{2}(2y-1)$$

then
$$0 < y - \frac{1}{2} + (x+1)^2 + (y-1)^2 - \mathrm{R}^2$$
$$0 < (x+1)^2 + y^2 - 2y + 1 + y - \frac{1}{2} - R^2$$
$$0 < (x+1)^2 + y^2 - y + \frac{1}{2} - R^2$$
$$0 < (x+1)^2 + (y - \frac{1}{2})^2 + \frac{1}{4} - R^2$$

Alternate Phrasing (3/3)

- The ***radial distance decision*** is whether

$$d_1 = (x+1)^2 + \left(y - \frac{1}{2}\right)^2 + \frac{1}{4} - R^2$$

  is positive or negative.

- The ***vertical distance decision*** is whether

$$d_2 = (x+1)^2 + \left(y - \frac{1}{2}\right)^2 - R^2$$

  is positive or negative; d_1 and d_2 are ¼ apart.

- The integer d_1 is positive only if $d_2 +$ ¼ is positive (except special case where $d_2 = 0$: remember you're using integers).

Incremental Computation Revisited (1/2)

- How can we compute the value of

$$f(x, y) = (x+1)^2 + \left(y - \frac{1}{2}\right)^2 - R^2$$

  at successive points? (vertical distance approach)

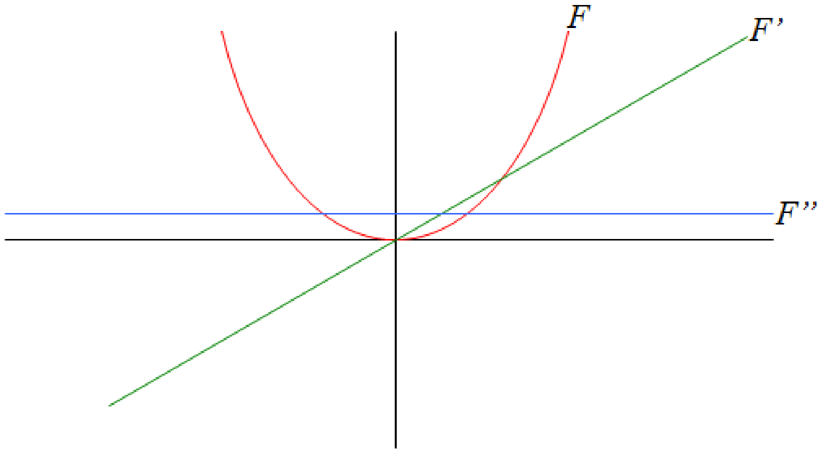
- Answer:
  - Note that $f(x+1, y) - f(x, y)$

    $= \Delta_E\ (x, y) = 2x + 3$
  - and that $f(x+1, y-1) - f(x, y)$

    $= \Delta_{SE}(x, y) = 2x - 2y + 5$

Incremental Computation (2/2)

- If we move E, update d = f(M) by adding $2x + 3$
- If we move SE, update d by adding $2x - 2y + 5$
- Forward differences of a 1st degree polynomial are constants and those of a 2nd degree polynomial are 1st degree polynomials
  - this "first order forward difference," like a partial derivative, is one degree lower



Second Differences (1/2)

- The function $\Delta_{\mathrm{E}}(x, y) = 2x + 3$ is linear, hence amenable to incremental computation:

$$\Delta_{\mathrm{E}}(x+1, y) - \Delta_{\mathrm{E}}(x, y) = 2$$
$$\Delta_{\mathrm{E}}(x+1, y-1) - \Delta_{\mathrm{E}}(x, y) = 2$$

- Similarly

$$\Delta_{\mathrm{SE}}(x+1, y) - \Delta_{\mathrm{SE}}(x, y) = 2$$
$$\Delta_{\mathrm{SE}}(x+1, y-1) - \Delta_{\mathrm{SE}}(x, y) = 4$$

Second Differences (2/2)

- For any step, can compute new $\Delta_{\mathrm{E}}(x, y)$ from old $\Delta_{\mathrm{E}}(x, y)$ by adding appropriate second constant increment – update delta terms as we move. This is also true of $\Delta_{\mathrm{SE}}(x, y)$.
- Having drawn pixel (a, b), decide location of new pixel at $(a + 1, b)$ or $(a + 1, b - 1)$, using previously computed $\Delta(a, b)$
- Having drawn new pixel, must update $\Delta(a, b)$ for next iteration; need to find either $\Delta(a + 1, b)$ *or* $\Delta(a + 1, b - 1)$ depending on pixel choice
- Must add $\Delta_E(a, b)$ or $\Delta_{SE}(a, b)$ to $\Delta(a, b)$
- So we…
  - Look at d to decide which to draw next, update x and y
  - Update d using $\Delta_E(a, b)$ *or* $\Delta_{SE}(a, b)$
  - Update each of $\Delta_E(a, b)$ *and* $\Delta_{SE}(a, b)$ for future use
  - Draw pixel

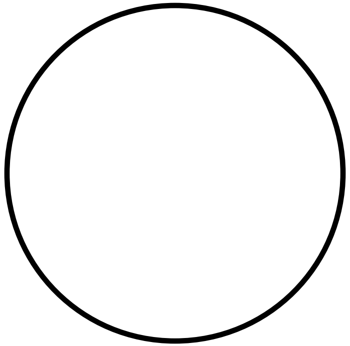
Midpoint Eighth Circle Algorithm

```
MidpointEighthCircle(R) { /* 1/8th of a circle w/ radius R */
    int x = 0, y = R;
    int deltaE   = 2 * x + 3;
    int deltaSE  = 2 * (x - y) + 5;
    float decision = (x + 1) * (x + 1) + (y - 0.5) * (y - 0.5) – R*R;
    WritePixel(x, y);

    while ( y > x ) {
        if (decision > 0) { // Move East
            x++; WritePixel(x, y);
            decision += deltaE;
            deltaE += 2; deltaSE += 2; // Update delta
        } else { // Move SouthEast
            y--; x++; WritePixel(x, y);
            decision += deltaSE;
            deltaE += 2; deltaSE += 4; // Update delta
        }
    }
}
```

Analysis

- Uses floats!
- 1 test, 3 or 4 additions per pixel
- Initialization can be improved
- Multiply everything by 4: No Floats!
  - Makes the components even, but sign of decision variable remains same



Questions

- Are we getting all pixels whose distance from the circle is less than ½?
- Why is $y > x$ the right criterion?
- What if it were an ellipse?